Object-Oriented Programming I (CSE 121)
Week # 03
(YouTube Lecture: 05 & 06)
Functions in C++ (User-defined)

Objective(s)

In this chapter, you will learn:

- User-defined functions syntax, categories, examples and benefits
- Function overloading rules and examples

A **function** is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. There are two types of function:

1. **Standard Library Functions:** Functions which are declared in the C++ header files such as ceil(x), pow(x, y), sqrt(x) etc.
2. **User-defined Functions:** Functions which are created by the programmer, so that he/she can use it many times.

In this lecture, we will focus mostly on user-defined functions.

C++ User-defined Function

- C++ allows the programmer to define their own function.
- A user-defined function groups code to perform a specific task and that group of code is given a name (identifier).
- When the function is invoked from any part of the program, it all executes the codes defined in the body of the function.

Benefits of user-defined functions in C++

- Functions make the code **reusable**. We can declare them once and use them multiple times.
- Functions make the program **easier** as each small task is divided into a function.
- Functions increase **readability**.

Declaring, Defining and Calling a Function

1. **Function declaration or prototype** – A function **declaration** tells the compiler about the function name, function parameters and return value's data type.

   Syntax: return_type function_name(parameter_list);

   Example: `int sum(int num1, int num2);`

   Parameter names are not important in function declaration only their type is required, so following is also valid declaration –

   `int sum(int, int);`

   Note: To create (often referred to as *declare*) a function, specify the name of the function, followed by parentheses **()**

2. **Function definition**: Writing the full body of function is known as defining a function.

Syntax:

```
return_type function_name(parameter_list) {
        //function body
    }
```

Example:

```
int sum(int num1, int num2) {
        return num1 + num2;
}
```

Explanation:

Return Type − It is the value that the functions return to the calling function after performing a specific task. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body − A group of statements that carry out a specific task.

3. **Function calling**: While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

Syntax: function_name(parameter_list);

Example: sum(num1, num2);

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

Parameters and Arguments

- Information can be passed to functions as a parameter. Parameters act as variables inside the function.
- Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

Syntax

```
void functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

The following example has a function that takes a `string` called **fname** as parameter. When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

parameterArgument.cpp

```cpp
#include<iostream>
using namespace std;

void myFunction(string);

int main()
{
    myFunction("Aleeza");
    myFunction("Esha");
    myFunction("Bulbul");
    return 0;
}
void myFunction(string fname){
    cout << fname << " Ahamed" << endl;
}
```

C:\Users\Bulbu

```
Aleeza Ahamed
Esha Ahamed
Bulbul Ahamed
```

When a **parameter** is passed to the function, it is called an **argument**. So, from the example above: `fname` is a **parameter**, while `Aleeza`, `Esha` and `Bulbul` are **arguments**.

> **Note:** If a user-defined function, such as myFunction() is declared after the main() function, **an error will occur**. It is because C++ works from top to bottom; which means that if the function is not declared above main(), the program is unaware of it:
>
> You will often see C++ programs that have function declaration above main(), and function definition below main(). This will make the code better organized and easier to read.

Return Values

The `void` keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as `int`, `string`, etc.) instead of `void`, and use the `return` keyword inside the function.

C++ function arguments and return values

For better understanding of arguments and return in functions, user-defined functions can be categorized as:

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and a return value

1. Function with no arguments and no return value

Function with no argument means the called function does not receive any data from calling function and function with no return value means calling function does not receive any data from the called function. So, there is no data transfer between calling and called function.

<u>Syntax:</u>

Function declaration	:	void function();
Function calling	:	function();
Function definition	:	void function() { statements; }

<u>Example:</u> Add two integer numbers

one.cpp

```
#include<iostream>
using namespace std;

void sum();//function declaration

int main(){
    sum();//function calling
    return 0;
}
//function definition
void sum(){
    int x, y, s;
    cout << "Value of x & y: ";
    cin >> x >> y;
        s = x + y;
    cout << "x + y = " << s;
}
```

C:\Users\Bulbul\Desktop\Functior

```
Value of x & y: 10 20
x + y = 30
Process returned 0 (0x0)
Press any key to continue.
```

2. Function with no arguments and a return value

Function with no arguments means called function does not receive any data from calling function and function with a return value means one result will be sent back to the calling function.

<u>Syntax:</u>

Function declaration	**:**	int function();
Function calling	**:**	function();
Function definition	**:**	int function() { statements; return x; }

<u>Example:</u> Add two integer numbers

two.cpp

```cpp
#include<iostream>
using namespace std;

int sum();//function declaration

int main(){
    int r = sum();//function calling
    cout << "x + y = " << r << endl;
    return 0;
}
//function definition
int sum(){
    int x, y, s;
    cout << "Value of x & y: ";
    cin >> x >> y;
        s = x + y;
    return s;
}
```

C:\Users\Bulbul\Desktop\Functior

```
Value of x & y: 10 20
x + y = 30
Process returned 0 (0x0)
Press any key to continue.
```

3. Function with arguments and no return value

Here function will accept data from the calling function as there are arguments, however, since there is no return type nothing will be returned to the calling program. So, it's a one-way type communication.

<u>Syntax:</u>

Function declaration	**:**	void function(int);
Function calling	**:**	function(x);
Function definition	**:**	void function(int x) { statements; }

<u>Example:</u> Add two integer numbers

three.cpp

```cpp
#include<iostream>
using namespace std;

void sum(int,int);//function declaration

int main(){
    int x, y;
    cout << "Value of x & y: ";
    cin >> x >> y;
    sum(x,y);//function calling
    return 0;
}
//function definition
void sum(int x, int y){
    int s;
    s = x + y;
    cout << "x + y = " << s << endl;
}
```

C:\Users\Bulbul\Desktop\Functior

```
Value of x & y: 10 20
x + y = 30
Process returned 0 (0x0)
Press any key to continue.
```

Note: Inside the function, you can add as many parameters as you want. Remember that when you are working with **multiple parameters**, the function call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

4. Function with arguments and a return value

Function with arguments and a return value means both the calling function and called function will receive data from each other. It's like a dual communication.

Syntax:

Function declaration	**:**	int function (int);
Function calling	**:**	function(x);
Function definition	**:**	int function(int x) { statements; return x; }

Example: Add two integer numbers

four.cpp

```
#include<iostream>
using namespace std;

int sum(int,int);//function declaration

int main(){
    int x, y, r;
    cout << "Value of x & y: ";
    cin >> x >> y;
    r = sum(x,y);//function calling
    cout << "x + y = " << r << endl;
    return 0;
}
//function definition
int sum(int x, int y){
    int s;
    s = x + y;
    return s;
}
```

C:\Users\Bulbul\Desktop\Functior

```
Value of x & y: 10 20
x + y = 30
Process returned 0 (0x0)
Press any key to continue.
```

Which method is better?

- All four programs above give the same output and all are technically correct program.
- There is no hard and fast rule on which method should be chosen.
- The particular method is chosen depending upon the situation and how you want to solve a problem.

C++ Function Overloading

Function overloading is defined as the process of having two or more function with the same name, but different in parameters. In function overloading, the function is redefined by using either different **types** of arguments or a different **number** of arguments or a different **sequence** of arguments. For example,

```
void largest();
int largest(int x, int y);
int largest(int x, int y, int z);
int largest(int x, int y, int z, int k);
```

Here, all 4 functions are overloaded functions.
Notice that the return types of all these 4 functions are not the same. Overloaded functions may or may not have different return types but they must have different arguments. For example,

```
int largest(int x, int y);
void largest(int x, int y);
```

Here, both functions have the same name, the same type, and the same number of arguments. Hence, the **compiler will throw an error**.

Program: Find the largest between/among 2, 3 and 4 numbers using function overloading

```
#include<iostream>
using namespace std;

void largest(); //function with no arguments
int largest(int x, int y); //function with two arguments
int largest(int x, int y, int z); //function with three arguments
int largest(int x, int y, int z, int k); //function with four arguments

int main(){
  int choice, r, r1, r2, x, y, z, k;
  largest();
  while(1) //1 means true
  {
    cout << "Enter your choice: ";
    cin >> choice;
    switch(choice){
    case 1:
      cout << "Enter the value of x & y: ";
      cin >> x >> y;
      r = largest(x,y);
      cout << "Largest = " << r << endl;
      break;
    case 2:
      cout << "Enter the value of x, y & z: ";
      cin >> x >> y >> z;
      r1 = largest(x,y,z);
      cout << "Largest = " << r1 << endl;
      break;
```

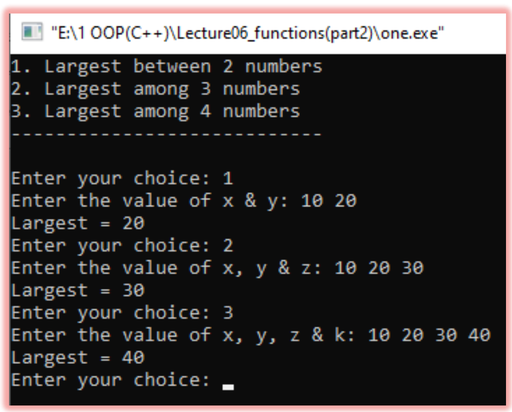
```
      case 3:
        cout << "Enter the value of x, y, z & k: ";
        cin >> x >> y >> z >> k;
        r2 = largest(x,y,z,k);
        cout << "Largest = " << r2 << endl;
        break;
      default:
        cout << "Invalid Choice. Please try again.\n";
      }
    }
  return 0;
}

//Program menu display
void largest(){
  cout << "1. Largest between 2 numbers\n";
  cout << "2. Largest among 3 numbers\n";
  cout << "3. Largest among 4 numbers\n";
  cout << "---------------------------\n\n";
}

//Largest between two numbers
int largest(int x, int y)
{
  int m = (x > y) ? x : y;
  return m;
}

//Largest among three numbers
int largest(int x, int y, int z)
{
  int m = (x > y && x > z) ? x :
      (y > z) ? y : z;
  return m;
}

//Largest among four numbers
int largest(int x, int y, int z, int k)
{
  int m = (x > y && x > z && x > k) ? x :
      (y > z && y > k) ? y :
      (z > k) ? z : k;
  return m;
}
```



Note: In C++, many standard library functions are overloaded. For example, the sqrt() function can take double, float, int, etc. as parameters. This is possible because the sqrt() function is overloaded in C++.

Formal and Actual Parameters

- The parameters that are defined in the function definition are called **Formal Parameters**.
- The parameters in the function call which are the actual values are called **Actual Parameters/arguments.**

Default Parameter Value

- You can also use a default parameter value, by using the equals sign (=).
- If we call the function without an argument, it uses the default value ("Bangladesh").

defaultParameter.cpp

```cpp
#include<iostream>
using namespace std;

void myFunction(string country = "Bangladesh");

int main()
{
    myFunction("Australia");
    myFunction();
    myFunction("Sweden");
    return 0;
}
void myFunction(string country){
    cout << country << endl;
}
```

C:\Users\Bu

```
Australia
Bangladesh
Sweden
```

Exercises:

1. Find the largest number between two numbers using four user-defined functions category.
2. Find the largest number between three numbers using four user-defined functions category.
3. Find the square root of a given number using four user-defined functions category.
4. Find the area of a circle using four user-defined functions category.
5. Find SUM and AVERAGE of two integer numbers using four user-defined functions category.
6. Check a given number whether it is prime or not using four user-defined functions category.

....